1. This problem is related to Vector processors. One common application in scientific fields is the inner product of two vectors. Consider the following loop that generates the inner product of two arrays with 64 elements:

   \[
   \text{sum} = 0.0;
   \text{for} \ (i=0; \ i<64; \ i++) \ \\
   \text{sum} = \text{sum} + A[i] \times B[i];
   \]

   Write vector code using RSIC-5 and RV64V instructions (as shown in textbook and in my examples). Assume \( x1 \) and \( x2 \) contain starting addresses of \( A \) and \( B \). Show how many convoys you need to complete them. Remember you will need to use a reduction operation that reduces an array into a single number (by adding individual elements of an array). You first store the result of the multiplication in a third vector and use reduction on the third vector.

2 (4.9 from textbook). Consider the following code, which multiplies two vectors that contain single-precision complex values (real and imaginary parts):

   \[
   \text{for} \ (i=0;\ i<300;\ i++) \ \\
   \text{c_re}[i] = \text{a_re}[i] \times \text{b_re}[i] - \text{a_im}[i] \times \text{b_im}[i]; \\
   \text{c_im}[i] = \text{a_re}[i] \times \text{b_im}[i] + \text{a_im}[i] \times \text{b_re}[i]; \\
   \]

   Write vector code using RSIC-5 and RV64V instructions. Assume registers \( x1 \) through \( x6 \) contain the starting addresses of arrays \( \text{a_re} \), \( \text{a_im} \), \( \text{b_re} \), \( \text{b_im} \), \( \text{c_re} \) and \( \text{c_im} \) respectively.

   Note that you need to use vector length register since you need to repeat the loop for 300 iterations.

3 (5.1 from textbook). A multicore SMT multiprocessor is illustrated in Figure 5.37 (shown below). Only the cache contents are shown. Each core has a single, private cache with coherence maintained using the snooping coherence protocol. Each cache is direct-mapped, with four lines, each holding 2 bytes (to simplify diagram). For further simplification, the whole line addresses in memory are shown in the address fields in the caches, where the tag would normally exist. The coherence states are denoted \( M \), \( S \), and \( I \) for Modified, Shared, and Invalid.

   For each part of this exercise, the initial cache and memory state are assumed to initially have the contents shown in Figure 5.37 (shown below). Each part of this exercise specifies a sequence of one or more CPU operations of the form:

   \( \text{Ccore#}: \ \text{R}, \ < \text{address}> \) for reads

   and

   \( \text{Ccore#}: \ \text{W}, \ < \text{address}> \ \leftarrow \ \text{<value written>} \) for writes.
For example,

C3: R, AC10  // Core C3 reads address AC10 (in hex)
C0: W, AC18 <-- 0018  // Core C0 write 0018 to address AC18 (again in hex)

Read and write operations are for 1 byte at a time. Show the resulting state (i.e., coherence state, tags, and data) of the caches and memory after the actions given below. Show only the cache lines that experience some state change; for example:

C0.L0: (I, AC20, 0001) indicates that line 0 in core 0 assumes an “invalid” coherence state (I), stores AC20 from the memory, and has data contents 0001. Furthermore, represent any changes to the memory state as M: < address > <-- value.

Different parts (a) through (g) do not depend on one another: assume the actions in all parts are applied to the initial cache and memory states.

a. C0: R, AC20
b. C0: W, AC20 <-- 80
c. C3: W, AC20 <-- 80
d. C1: R, AC10
e. C0: W, AC08 <-- 48
f. C0: W, AC30 <-- 78
g. C3: W, AC30 <-- 78

4 (5.2 from textbook). The performance of a snooping cache-coherent multiprocessor depends on many detailed implementation issues that determine how quickly a cache responds with data in an exclusive or M state block. In some implementations, a processor read miss to a cache block that is exclusive in another processor's cache is faster than a miss to a block in memory. This is because caches are smaller, and thus faster, than main memory. Conversely, in some implementations,
misses satisfied by memory are faster than those satisfied by caches. This is because caches are generally optimized for “front side” or CPU references, rather than “back side” or snooping accesses. For the multiprocessor illustrated in Figure 5.37, consider the execution of a sequence of operations on a single processor core where

- read and write hits generate no stall cycles;
- read and write misses generate \( N_{\text{miss}} \) and \( N_{\text{act}} \) stall cycles if satisfied by memory and cache, respectively;
- write hits that generate an invalidate incur \( N_{\text{invalidate}} \) stall cycles; and
- a write-back of a block, either due to a conflict or another processor's request to an exclusive block, incurs an additional \( N_{\text{writeback}} \) stall cycles.

Consider two implementations with different performance characteristics summarized in Figure 5.38 (shown below)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Implementation 1 Cycles</th>
<th>Implementation 2 Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_{\text{memory}} )</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>( N_{\text{cache}} )</td>
<td>40</td>
<td>130</td>
</tr>
<tr>
<td>( N_{\text{invalidate}} )</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>( N_{\text{writeback}} )</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

To observe how these cycle values are used, we illustrate how the following sequence of operations, assuming the initial caches' states in Figure 5.37 (from previous problem), behave under implementation 1.

C1: R, AC10
C3: R, AC10

For simplicity, assume that the second operation begins after the first completes, even though they are on different processor cores.

For Implementation 1,

- the first read generates 50 stall cycles because the read is satisfied by C0's cache: C1 stalls for 40 cycles while it waits for the block, and C0 stalls for 10 cycles while it writes the block back to memory in response to C1's request; and
- the second read by C3 generates 100 stall cycles because its miss is satisfied by memory.

Therefore this sequence generates a total of 150 stall cycles.
For the following sequences of operations, how many stall cycles are generated by each implementation?

a. 
   C0: R, AC20
   C0: R, AC28
   C0: R, AC30

b. 
   C0: R, AC00
   C0: W, AC08 <-- 48
   C0: W, AC30 <-- 78

c. 
   C1: R, AC20
   C1: R, AC28
   C1: R, AC30

d. 
   C1: R, AC00
   C1: W, AC08 <-- 48
   C1: W, AC30 <-- 78